ESIR SPP – TD2 & 3

Exercise 1:

One of your colleagues, who hasn't followed the SPP module at ESIR, comes up with the following design for a banking application.

```
public class Account {

  private int amount = 0 ;

  public Account(int startAmount) {
    amount = startAmount ;
  }

  synchronized public void withdraw(int amountToWithdraw) {
    amount -= amountToWithdraw;
  } // endMethod withdraw

  synchronized public void deposit(int amountToDeposit) {
    amount += amountToDeposit;
  } // endMethod deposit

  synchronized public void transferTo(Account otherAccount, int transfer) {
    this         .withdraw(transfer);
    otherAccount.deposit (transfer);
  } // endMethod transferTo
} // endClass Account
```

1) Can you spot any problem if this code? Will it run seamlessly in a multithreaded context? Propose a scenario that illustrates this problem.

2) How can you solve the problem?

Exercise 2:

While looking on the Internet for an implementation of semaphores that does not use busy waiting, you unearth the following algorithm.

```
lock l1, l2 // l2 initialised as taken, l1 as free
int counter = start_value // counter can be negative during execution

method up() is
 l1.lock()
 counter++
 if (counter<=0) {
   l2.unlock() // others are waiting
 }
 l1.unlock()
end

method down() is
 l1.lock()
 counter--
 l1.unlock()
 if (counter<0) {
   l2.lock() // not enough permits
 }
end
```

1) What do you think are the intended use of the locks l1 and l2, respectively.

2) This algorithm does not work. Why? Find a scenario that illustrates this problem. Can it be solved without busy waiting using locks only?

3) Using the same general idea as above, propose an algorithm that implements a semaphore using a monitor instead of locks, and avoids the problem discussed in question 2.

Exercise 3:

Return to the solution of the producer-consumer problem that uses locks. Propose a version that now uses a monitor.

Exercise 4:

None of the solutions to the producer-consumer problem we have seen so far allow for parallel access to the queue of items (be it in read/read, read/write, write/write mode). One of your colleagues proposes the following implementation, where the actual accesses to the queue (in writing for the producer, or reading for the consumer) occur outside of the mutual exclusion.

```
lock l1, l2
int in  = 0 // beginning of queue, where the next insertion should occur
int out = 0 // end of queue, where the next consumption should occur
data[] queue = new data[MAX]

method produce(x) is
 l1.lock()
 cell = in
 in = (in + 1)%MAX
 l1.unlock()
 data[cell] = x
end

method consume() is
 l2.lock()
 cell = out
 out = (out + 1)%MAX
 l2.unlock()
 result = data[cell]
 return result
end
```

1) What are the flaws in this code?

2) How would you solve these problems with two semaphores, while maintaining the concurrent accesses to the queue advocated by your colleague?